FUBAR: Flow Utility Based Routing

Nikola Gvozdiev, Brad Karp, Mark Handley
University College London, *initial.lastname@cs.ucl.ac.uk*

ABSTRACT

We present FUBAR, a system that reduces congestion and maximizes the utility of the entire network by installing new routes and changing the traffic load on existing ones. FUBAR works offline to periodically adjust the distribution of traffic on paths. It requires neither changes to end hosts nor precise prior knowledge of the traffic matrix. We demonstrate that even in the presence of traffic from all network devices to all other devices, FUBAR can optimize a real-world core-level network in a matter of minutes.

Categories and Subject Descriptors

C.2.1 [**Computer-Communictaion Networks**]: Network Architecture and Design

General Terms: Algorithms; Design

1. INTRODUCTION

What should be the role of the routing system in a large ISP or enterprise network? Traditionally, it was all about ensuring connectivity, with cost metrics used as a crude way to spread load. Equal-cost multipath routing [3] added limited capability to load-balance, and MPLS-TE [4] provided additional ability to tweak traffic patterns to mitigate congestion. More recently, OpenFlow and Software Defined Networking [10] have provided a much more flexible data plane. But how should a routing system resolve applications' competing, qualitatively different demands?

Large networks with sufficiently many users tend to have relatively stable traffic matrices—demand between each entry point and exit point does not vary very quickly. At the same time, the demand from each traffic aggregate is elastic, but not infinitely so. Streaming video, for example, can switch between different data rates, but once it reaches sufficient quality, bitrate becomes bounded. Even file transfer, which ought in principle to be infinitely elastic, is generally not in reality, due to access link capacity, receive window bounds and I/O limitations. In fact much web traffic is latency bounded, as it finishes while still in slowstart. The job of the routing system should be to satisfy all traffic aggregates, subject to there actually being enough capacity available in the network. But bandwidth isn't the only factor to be considered. Real-time flows have much firmer latency requirements—there's no point in giving arbitrary bandwidth to videoconferencing traffic if doing so involves routing it over a high-delay path. The routing system should instead try to maximize the utility of the traffic flowing over it, where utility depends on both delay and throughput. This is a very different problem than that solved by traditional routing protocols.

In this paper we describe Flow-Utility Based Routing, a new centralized routing mechanism. Based on information from switches, FUBAR measures the network's traffic matrix and determines how to route the aggregates that comprise that matrix so as to maximize utility for the network's users. Not all traffic is treated equally; real-time traffic must be routed over lower-delay paths, and even bulk aggregates are routed over the lowest-delay path if there is sufficient capacity. If there is not sufficient capacity, FUBAR splits an aggregate, progressively moving traffic onto less preferred paths until there is no persistent congestion in the network, or no further gains in utility can be achieved.

Characterizing the goal of routing in this way makes it into a difficult problem. The pieces of the puzzle are:

- How to predict the utility of an aggregate, given a traffic class and a certain amount of bandwidth and delay?
- How to predict how multiple aggregates will share a link?
- Given these, how to choose a set of possible paths for each aggregate, including paths that avoid congested hot-spots?
- Finally, how to split aggregates between multiple paths in such a way that overall utility is maximized?

If we can satisfy all of these goals, we will also achieve a network with no persistent congestion given sufficient provisioning, or one in which there are no severely congested hotspots if the network is under-provisioned.

To achieve these goals, we make a number of gross simplifications. We start with a very crude utility model and fill in the model's parameters based on measurements of how the traffic actually responds. We classify traffic with crude heuristics supplemented by operator knowledge when that is available. We also use a fairly simplistic model of how traffic aggregates share a congested link. We believe these rough approximations capture the essence of how real traffic behaves. Crucially, they allow us to reason about and choose between options for routing, and require little or no operator input. Current routing systems do not even take such limited information into account; if they perform well, it is more due to luck and manual tuning by operators who rarely know the real demands of the traffic they are engineering.

Given these models, when the network is congested, we

This work was funded by EU FP7 grant #317756 "Trilogy 2"

HotNets-XIII, October 27–28, 2014, Los Angeles, CA, USA

ACM 978-1-4503-3256-9/14/10 ...$15.00
http://dx.doi.org/10.1145/2670518.2673886.

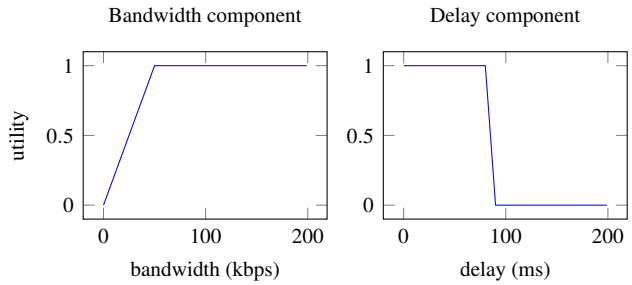

Figure 1: Real-time flow utility function components

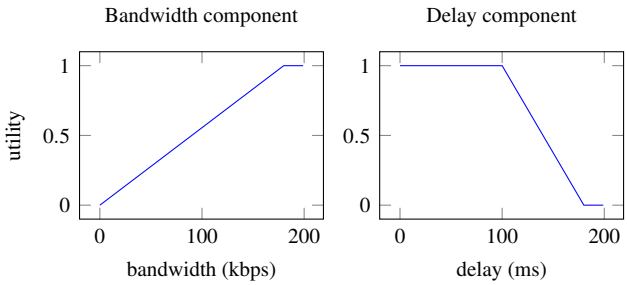

Figure 2: Bulk transfer flow utility function components

must generate paths that provide alternatives to the default low-delay paths. Then we must determine the optimal mapping of sub-aggregates to paths. This problem is NP-hard, but given the limitations of the models, there is little point in being too concerned with always finding an optimal solution. We care only about finding a good solution, and FUBAR performs sufficiently well in this respect to provide very substantial gains in utility over conventional shortest-path routing. Moreover, by alleviating congestion, FUBAR avoids building long queues in the network, even when operating at high network utilization.

2. SYSTEM DESIGN

The four components needed to implement FUBAR are:

- A way to measure the traffic matrix of a network.
- A method to approximate the utility of an individual flow.
- A method to estimate the bandwidth a flow will achieve given a defined path through the network in the presence of all the other flows in the network.
- A method to derive policy compliant paths for each flow, including the lowest delay path and others that avoid hotspots.
- An optimization algorithm to find the best way to split each aggregate of flows between the multiple policy compliant paths so as to maximize the overall utility of the network.

We will address each of these in turn.

2.1 Traffic Matrix

Traditionally, measuring the traffic matrix was difficult because routers did not keep sufficient counters. In an SDN network, the SDN controller is involved in setting up flows, so measuring the traffic matrix is much simpler.

FUBAR needs periodic per-aggregate bandwidth measurements and approximate flow counts for each aggregate. We believe that in real-world deployments FUBAR will be separate from the SDN controller and the measurements required will be taken hierarchically.

2.2 Utility

To capture how useful a flow is to the user application we borrow Shenker's concept of *utility* [11], but extend it to be a function of both bandwidth and delay. In FUBAR each flow is associated with a *utility function* which provides a mapping from bandwidth and delay to a single unitless real number in the range $[0-1]$. A utility close to 0 means that the flow does not accomplish any useful work. This may be because it currently does not have enough bandwidth or because the path that it takes through the network has too high a delay. Conversely, a utility close to 1 means that providing additional bandwidth or reducing the flow's delay are not likely to render it any more useful to the application.

The function itself defines a surface whose shape is very application dependent. Our utility metric consists of a bandwidth component and a delay component that are multiplied together to form the final utility. Examples of the two components can be seen in Figures 1 and 2.

For most applications a default delay curve can be used that slowly decays to zero as delay increases to a few seconds. Interactive applications tend to be more delay sensitive, so the operator can specify a non-default delay curve for flows to a certain port or from a particular server.

Our goal is to minimize the need for operator knowledge, so we rely on continuous traffic measurements to scale the bandwidth component as needed. We can infer the inflection point of the bandwidth curve when an aggregate is using an uncongested path and fails to utilize it. Our solution will also work with any non-linear increasing function for either bandwidth or delay, but for simplicity we chose those shapes because they are defined by the fewest points.

Figure 1 shows an example utility function of an interactive flow. At $0\,kbps$ the flow gets no bandwidth and is therefore not useful—its utility is 0. The utility grows as it gets more bandwidth and maxes out at $50\,kbps$, after which even if the flow is given more capacity the application is unable to make use of it. It can also be seen that this flow is delay sensitive—if the delay it experiences grows above $100\,ms$ the overall utility will drop to 0 (the two components are multiplied).

Figure 2 shows a bulk data transfer flow that can tolerate relatively large variations in delay, but requires more bandwidth. Conventional wisdom is that a single TCP flow with enough data to send is never satisfied—it will consume the entire capacity of the network and its bandwidth "peak" will be the bandwidth of the bottleneck link in its path. In real networks most TCP flows do not behave this way, especially in the backbone. The bandwidth demands of large transfers tend to be limited by the application itself (*e.g.,* the data rate of a compressed YouTube or Netflix video). Small transfers, on the other hand, are so small that they barely exit slow start. Even software downloads tend to be constrained in customers' access networks rather than in the backbone. When

we look at multiplexed aggregates our preliminary measurements show that it is almost always possible to define an upper bound on the bandwidth requirement at any instant.

2.3 Traffic Model

For any particular allocation of flows to paths, we need to know how the flows will occupy the network so that we can estimate their bandwidth and hence their utility.

If there is at least one congested link we are faced with predicting how its capacity will be split among flows that cross it. Each congested link truncates the demands of flows that traverse it, so affects the distribution of flows on other congested links.

To estimate the traffic distribution we use a TCP-like traffic flow model. The model assumes a congested flow's throughput is inversely proportional to its RTT, and that congestion control algorithms in use are "compatible" in the sense that they co-exist reasonably gracefully with TCP traffic.

We imagine the network as a series of empty pipes. We fill them by having each flow grow at a rate inversely proportional to its RTT. A flow can stop growing either because it satisfies its demand (obtained from the peak of the bandwidth component of the utility function) or because there is no more room to grow because a link along its path has become congested. In practice we don't deal with individual flows, but with *bundles* of flows that share the same entry point, exit point, traffic class, and path through the network. The algorithm proceeds in steps, congesting a link or satisfying a bundle at each step until each bundle is either congested or has its demands met.

This flow model is simple enough to run quickly, yet sufficient to provide us with a back-of-the-envelope estimate of what bandwidth each flow can expect to get given a path assignment. We use this flow model as the building block of our optimization algorithm outlined in the next section.

2.4 Choosing Paths

We want to be able to take an aggregate of flows that share a source, destination and traffic class, and to split this into bundles of flows that are routed over different paths through the network. This will allow us to progressively offload an aggregate from its lowest delay path onto higher delay but less congested paths, so long as doing so increases utility.

The default path for an aggregate is easy to find—it's simply the lowest delay path, as if that is uncongested it will give the best utility. However, as that path becomes congested, we want to consider offloading part of the aggregate onto other paths. An optimal algorithm would need to consider all the possible policy-compliant paths between that source and destination. This is clearly computationally infeasible, so we need to wisely choose paths to add as alternatives.

Unfortunately, the choice of a good alternative path depends not only on the topology but also on which links are congested, as we want to offload traffic onto uncongested paths. This leads to a catch 22: we cannot choose paths without knowing congestion, but we can't know congestion levels without choosing paths for the traffic to use.

Our solution to this dilemma is to take an iterative approach. We start with only the lowest delay path in the path set for an aggregate and run the traffic model. This will predict where congestion will occur. If there is no congestion, we're done.

If not, we add new paths to the path set for any aggregate that experiences congestion. The algorithm queries a path generator to find three alternative different policy-compliant paths not currently in the path set for that aggregate:

1. A global path: the lowest delay path that avoids all congested links, regardless of whether they are currently used by this aggregate.
2. A local path: the lowest delay path that avoids all congested links that are being used by the congested aggregate.
3. A link-local path: the lowest delay path that simply avoids the most congested link used by the aggregate.

The global path, if one exists, is the alternative giving maximum additional incremental capacity as it is uncongested, but it may have high delay. The link-local path is the lowest delay alternative, but may already be congested. The local path offers a good middle ground. We tried different approaches and found this particular choice of three paths to be the best tradeoff between speed and solution quality.

The algorithm iterates, finding the best way to split each aggregate across the paths in the path set, as described in the next section, then adding more paths to the path set of any aggregates still congested, and so on, until either no improvement is found or all aggregates are uncongested. In our experiments we usually find that three or four iterations are sufficient, so we end up with approximately ten to fifteen paths in the path set for each aggregate.

2.5 Flow Allocation

Given a network topology and a set of aggregate flows, plus for each aggregate, a utility function and a path set, we've finally reached the heart of the problem: how to split each aggregate across paths so as to maximize overall utility. An optimal solution is NP-hard, but given all the other approximations we have made to get this far, optimality is not of great importance. What we want is simply a good solution.

For this we use a polynomial-time heuristic. The main intuition is that at first flows are allocated on low delay paths and are gradually moved to higher delay paths until congestion disappears. The algorithm proceeds in steps, increasing utility at each step until no progress can be made. This greedy approach guarantees termination, but it can converge to a local optimum. We will come back later to what we do when it does this.

Initially all flows for each aggregate are allocated to the shortest delay path (line 1 from Listing 1). The traffic model is evaluated at this point and if there is no congestion the solution is optimal. In this case all flows have their bandwidth demand satisfied and are on the shortest policy-compliant delay path, which results in maximal attainable overall utility.

Listing 1: Main entry point

```
1  move all flows to lowest-delay path in aggregate
2  while there are congested links do
3  |   progress_made ⟵ false
4  |   links ⟵ all congested links
5  |   sort links by oversubscription
6  |   for link in links do
7  |   |   progress_made ⟵ step(link)
8  |   |   if progress_made then
9  |   |   |   break
10 |   if not progress_made then
11 |   |   break
```

If there is congestion the algorithm will try to alleviate it by moving some flows of each aggregate away. It looks at congested links one at a time, starting with links where a change is likely to result in the best utility gain—the ones that are the most oversubscribed (lines 4-5 from Listing 1).

Focusing on a single congested link (Listing 2), the algorithm goes over all aggregates crossing that link; for each it estimates the gain of moving some or all of the flows away. For a given flow path there are two main choices to be made—how many flows to evict and where to move the flows to.

The number of flows to move (N) depends on how large the aggregate is in terms of traffic volume. Small aggregates are moved in their entirety because they are unlikely to have a big impact on the final solution. For large aggregates there is a tradeoff between speed and utility—the more flows are moved at a time the faster the algorithm will converge, but the lower the overall utility of the solution will be.

Flows will have to be moved to an alternate path that excludes the congested link. As described in Section 2.4, the algorithm obtains three alternative paths (lines 4-6 from Listing 2).

At each step all three new alternatives are tested for each aggregate that crosses the congested link (lines 7-9 from Listing 2). When an aggregate is tested, N flows are removed from the original path and added to the alternative. The traffic model is evaluated to estimate the utility of the new solution. The best move is committed (line 12 of Listing 2). At the end of the step, if no move improves utility, no progress can be made and the algorithm terminates.

Escaping local optima

The flow allocation problem is not convex and it is possible for the greedy algorithm above to get stuck in a local optimum. In this case no progress can be made, even though there is still congestion in the network. There is a simple tweak that can help us escape—when the algorithm gets stuck we can try to move larger and larger numbers of flows. In particular while we are in a local optimum we can tweak line 3 from Listing 2 to progressively give more and more flows. This will help us explore the state space more aggressively. This algorithm is motivated by simulated annealing [9], but we have found it gives similar results in a much shorter time than a naive simulated annealing solution.

Listing 2: $step(link)$ – a single step of the algorithm

```
   input  : link – a congested link
   output : true if progress was made, false otherwise
1  U_init ⟵ run_model()
2  for flow_path in all flow paths that go over link do
3  |   N ⟵ fraction of flows in flow_path's aggregate
4  |   P_global ⟵ find global uncongested path
5  |   P_local ⟵ find local uncongested path
6  |   P_link-local ⟵ find link-local uncongested path
7  |   test utility of moving N flows to P_global
8  |   test utility of moving N flows to P_local
9  |   test utility of moving N flows to P_link-local
10 if best is still empty then
11 |   return false
12 commit the best utility change
13 return true
```

It is also possible that even though there is still congestion in the network we have reached the global optimum. In this case the algorithm will give up after having tried to move even large aggregates in their entirety and failing to improve overall utility.

3. EVALUATION

To better understand FUBAR's potential, we set out to answer the following questions:

- Is it computationally tractable for real-world deployments, at least for an offline system?
- How good are the routing solutions it finds?
- Does it manage to alleviate congestion?
- Is it able to prioritize some flows, while still retaining good overall performance?
- How do the utility functions affect performance?

To answer those questions we run FUBAR using Hurricane Electric's core topology [1]. This contains 31 POP nodes and 56 inter-POP links. For each of all 961 aggregates we randomly pick either a real-time utility function or a bulk-transfer one. To reflect real-world traffic we also add a 2% probability of there being a large aggregate using a file transfer utility function with a higher max bandwidth (1 or 2 Mbps). Delay utility functions are as shown in Figures 1 and 2. We examine two main cases:

Provisioned: each link of the topology has a capacity of 100 Mbps. This is enough capacity to make it possible to alleviate congestion, but not enough capacity for every flow to be satisfied on its shortest path

Underprovisioned: each link of the topology has a capacity of 75 Mbps. In this case there is not enough capacity to completely eliminate congestion.

Figures 3 and 4 present a single run of FUBAR in each of these two cases running on a 1.3 GHz Core i5. The current implementation is single-threaded and implemented in Java. The graphs show how the algorithm progressively optimizes the computed solution in real-time. Utility values are predicted utilities based on the traffic model described in 2.3.

The leftmost graph shows how utility evolves over time.

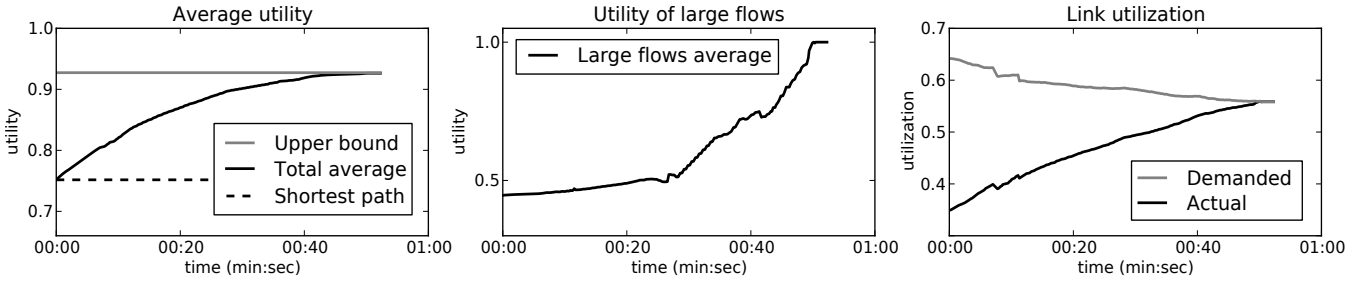

Figure 3: A run in the provisioned case

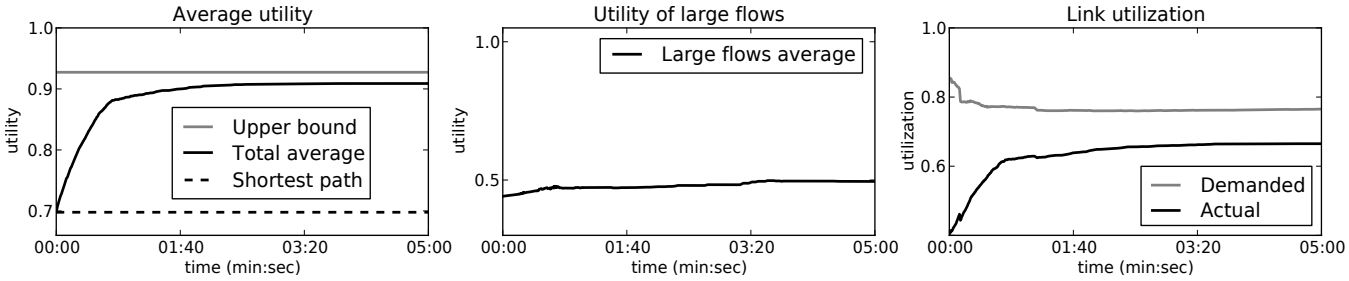

Figure 4: A run in the underprovisioned case

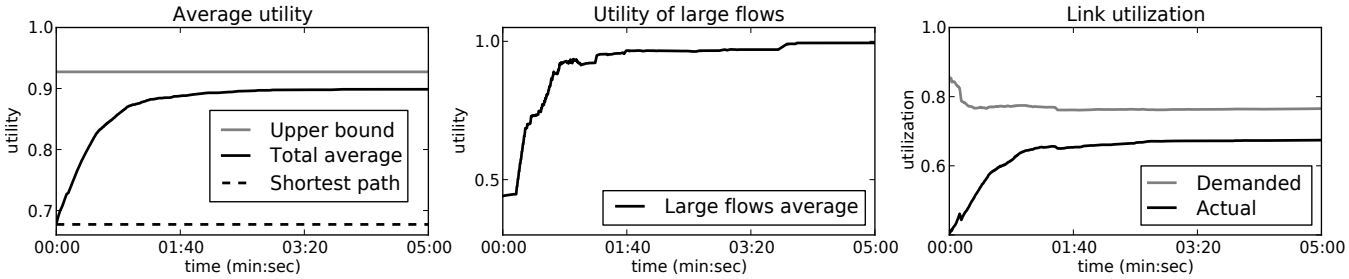

Figure 5: A run in the underprovisioned case with large flows prioritized

The "total average" is the overall utility of the network—the average of utilities of all aggregates, weighted by number of flows in the aggregate. The other two lines are for reference. To produce the "upper bound" curve we isolate an aggregate by removing all other aggregates from the network and determine what the single aggregate's utility would be if there were no other traffic. We repeat this for each aggregate and then take the mean. The "shortest path" line shows what utility would be if all the traffic takes its shortest path through the network.

The second graph on each row shows the utility of large flows, as these are harder to provide for and put disproportionate strain on the network.

In the right graphs we show link utilization. The "actual utilization" curve represents total used capacity divided by total network capacity[1]. The "demanded utilization" curve shows total demand[2] divided by total network capacity. Demanded utilization decreases as the optimization runs because total network capacity increases as more links are brought into play. If the two curves meet, demand has been satisfied.

[1]total network capacity is the sum of capacities across all links *that are used.*

[2]this is what the flows would have liked to get, again on links that are used

Running time. The running times of both cases are within the five minute limit for an offline system. In the provisioned case FUBAR finds a solution in less than a minute, whereas it takes about five minutes to reach an optimum when underprovisioned. In the latter case, the algorithm spreads out traffic, lightly congesting more and more links, taking longer to test moves over each one of them. Eventually there isn't a move that can improve utility, and the algorithm terminates.

Solution quality. In both cases FUBAR improves significantly on traditional shortest-path routing. This is expected as it starts by putting all flows on the shortest paths and improves from there on. Thus shortest-path routing is a lower bound for total utility. In the provisioned case FUBAR closely approaches the upper bound. In the underprovisioned case, although FUBAR improves utility by over 30%, the upper bound is unreachable.

It is also interesting to observe the utility of aggregates with large flows. By default all flows are treated equally regardless of their volume. It is much easier to initially gain utility by moving the small flows as provisioning for them results in a quick utility boost. In Figure 3, only after most small flows have been optimized do chunks of large flows start being moved; eventually all flows are optimized.

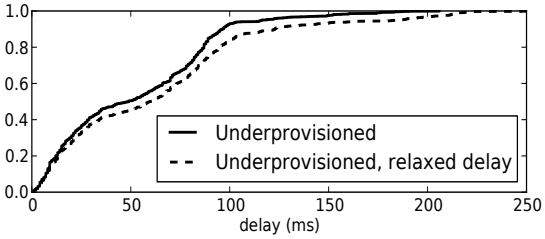

Figure 6: Increase in delay as result of relaxing delay restrictions

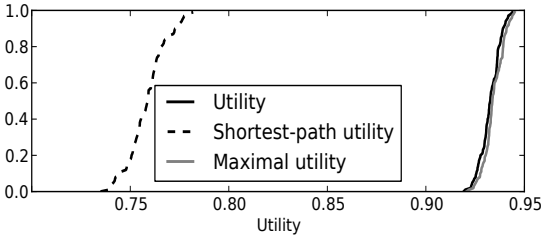

Figure 7: A CDF of 100 runs of the provisioned case

The observation that large flows are hard to optimize is also evident from Figure 4 where large flows are sacrificed to better accommodate smaller ones when resources are scarce. Thus the final utility of large flows is significantly lower than the global one.

Avoiding congestion. Minimizing congestion not only helps applications, but it also makes the network more predictable, as queue sizes are minimized. If there is enough capacity available we manage to eliminate congestion as is evident from the link utilization graph in Figure 3. When the two curves meet the capacity that is consumed by flows equals the demand of those flows and the network is not congested.

In the underprovisioned case link utilization improves by 80%, but in the end there is still congestion in the network (as evident from the gap). It would be possible to drive utilization even higher by saturating the network with large flows, starving smaller ones, but instead we try to improve utility for all participants.

Prioritization. Figure 5 shows a single run of the underprovisioned case when priority is given to large flows by increasing their weighting when computing the network utility. Now the utility of the large flows grows faster and is able to reach its peak. This comes with a slight increase of link usage, since larger flows occupy more network resources. The increase is not more because the number of small aggregates greatly outnumbers the number of large aggregates.

Another effect is the slower growth of overall utility because prioritizing large flows does initially hurt the numerous smaller ones. In the end, overall utility has not changed a great deal—the 1% reduction in small flow utility is offset by the gain in utility of larger ones.

Delay changes. To see the effect of the delay part of the utility function, we ran the underprovisioned case with small flows using double the delay parameter from Figures 1 and 2. Overall the results (not shown) look very similar to the Figure 4 curve, though utility and utilization increase a little as longer paths are no longer penalized. Overall run time is also increased slightly since the algorithm takes more time to explore now-viable alternatives.

It is also interesting to see how the overall delay distribution changes. In Figure 6 we present CDFs of the delays of all flows in the network in the original and the relaxed delay case. It is evident that even though overall utility has increased, paths have lengthened, with a median delay increase of about $10\,ms$ and tail-end increase of about $50\,ms$. For real-time traffic, every millisecond matters; the ability to trade utilization for delay by tuning a single parameter is a very useful one for network operators to have at their disposal.

Repeatability. The results so far have only shown experiments with a single run of the algorithm. Are the results stable across different traffic matrices? We ran 100 passes of the provisioned case with the same topology, but with different random seeds for choosing the traffic matrices. We present the results in Figure 7 as a CDF. We show curves for FUBAR's utility in comparison to shortest-path and the upper-bound maximal utility. In all runs FUBAR's performance closely approached the theoretical limit, as in Figure 3.

4. RELATED WORK

MPLS-TE [4] provides a flexible substrate for placing flows on paths, and Constrained Shortest-Path First (CSPF) [5] places flows on MPLS-TE paths that meet operator-pre-defined constraints. CSPF does not optimize global utility across all flows, however, as does FUBAR.

There is a rich literature on traffic engineering for IP networks [2, 6–8, 12]. While several of these prior approaches maximize global utility, they define utility only in terms of throughput and/or minimization of maximum utilization. Unlike FUBAR, none weighs the combined effects of delay and throughput on applications' flows. B4 [7] and SWAN [6] support prioritization of different traffic classes, but priority alone cannot capture the non-convexity of delay in applications' utility functions.

Our goals in FUBAR are related to Winstein's [13], but our approach differs. Rather than modifying end-hosts' congestion control, we modify resource allocation within the network. We believe this approach to be easier to deploy.

5. CONCLUSION

We have presented FUBAR, a system that provides utility-based routing for flow aggregates in large ISPs or corporate networks. FUBAR performs unequal-cost and weighted multipath routing, taking into account flows' measured bandwidth demands and configured delay preferences. It can operate networks at high utilization while avoiding congested hotspots if sufficient capacity is available, or diffusing hotspots if insufficient capacity is available. It is intended to be used as an offline controller in SDN or MPLS networks, in conjunction with an online controller to actually admit flows to the paths that have been computed.

6. REFERENCES

[1] Hurricane Electric IP Transit Network. *https://www.he.net*.

[2] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *ACM SIGCOMM 2003*.

[3] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. *RFC 2992*, 2000.

[4] D. Awduche and J. Malcolm. Requirements for Traffic Engineering Over MPLS. *RFC 2702*, 2009.

[5] B. Davie and A. Farrel. *MPLS: Next Steps*. Morgan Kauffmann, 2008.

[6] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM 2013*.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM 2013*.

[8] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM 2005*.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[11] S. Shenker. Fundamental design issues for the future internet. *IEEE JSAC*, 13(7):1176–1188, Sept. 1995.

[12] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg. COPE: Traffic engineering in dynamic networks. In *ACM SIGCOMM 2006*.

[13] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated congestion control. In *ACM SIGCOMM 2013*.